Performance of Weak Consistency Schemes on the DEC Alpha

Performance of Weak Consistency Schemes on the DEC Alpha

Tim Harris and Nigel Topham [a] *

[a]Department of Computer Science, University of Edinburgh,
The King's Buildings, Edinburgh, Scotland, EH9 3JZ

The performance of a shared memory multiprocessor is largely dependent upon the model of shared memory that is presented to the user. Where the first such machines typically supported a very powerful model of shared memory, that of *sequential consistency*, more recent designs have often benefited from the use of weaker memory models. However, there has been little standardisation between these weak models, and little practical work has been done to outline their possible implementations and performance. In the following we consider the weakly consistent model of shared memory which is supported by the DEC Alpha, which promises to be a common building block for such multiprocessors in the future. We suggest possible implementations of the model, using both hardware and software techniques, and consider the subsequent performance of these schemes on a variety of applications.

1. Introduction

Shared memory has proven to be a powerful model of parallel computation, but it is often a challenge to implement the model efficiently given the discrepancy between the fast performance of CPUs and the slow performance of memory systems in current technologies. Traditionally shared memory multiprocessors supported strongly consistent models of shared memory, in the sense that the ordering of instructions which access the shared memory was as close to that of a uniprocessor as possible. The most common such scheme is *sequential consistency*, where the ordering of multiprocessor instructions is defined such that (i) all the operations of a single processor are ordered in sequential order, and (ii) the operations of the entire machine are ordered as some interleaving of these sequential threads [1]. More recently, multiprocessor architects have realised that the efficiency of a shared memory implementation can be substantially improved by the use of a weaker model of consistency, where instructions may be reordered in a more liberal way which may contradict the premises of sequential consistency. A large variety of such models have been suggested and analysed, but few have been implemented in practice yet.

In this paper we consider the model of shared memory that is supported by the DEC Alpha microprocessor. Given that this chip will likely form the basis of a variety of future multiprocessors, the memory model it supports may eventually become a *de facto* standard among weakly consistent models. We suggest possible implementations for the Alpha model when used on a cache-based shared memory machines. In particular, we outline a possible hardware coherency scheme and a family of software schemes which are

*This work was funded by the the European Community ESPRIT project SHIPS, number P6253.

designed for use on a typical shared memory architecture. We then provide experimental evidence as to the performance of these schemes on a set of common applications. These results have been generated through use of a discrete event simulation which takes as input a parallel address trace resulting from the execution of three applications from the *Perfect Club* benchmark suite [2].

We show that with some applications one can expect very similar performance between Software and Hardware schemes. However, as the Epoch size and the amount of reuse in an application decreases the performance of a Software scheme will fall off steeply in comparison to that of a hardware scheme. We also observe that the synchronisation primitive of the Alpha memory model provides a useful abstraction for defining the Epochs necessary for the use of a Software Coherency scheme.

2. Architectural Assumptions

We assume a typical design for a parallel shared memory machine with a modest number of processors. In particular, we assume that all our processors share access to one memory sub-system, and hence our design is likely not scalable past a modest number of processors. However, the large majority of shared memory multiprocessors in use today also share this limitation. Our results are primarily presented within the context of a UMA architecture (Uniform Memory Access), i.e. we assume constant cost memory access throughout most of our analysis. This is primarily to simplify the understanding of our results, as our goal is not to assess the performance of our memory system as much as it is to evaluate the effects of that memory system performance within the context of a weakly consistent model.

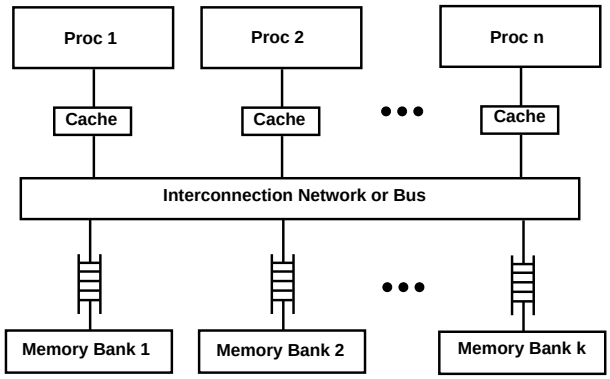

Figure 1. Assumed Architecture.

A simple schematic of the assumed architecture can be seen in figure 1. It shows a multiprocessor, where each processor has a local cache, and where all processors may access (over a crossbar switch or other interconnection network) an interleaved memory

system with multiple memory banks. Each memory bank has its own queue and we assume requests are not reordered within these queues. Such a memory system is designed to allow pipelining of memory requests. The cache sizes will be varied in our analysis, as will the number of processors and the implementation of our coherency scheme. Our cache is direct mapped and has a "fast" access time of five cycles, whereas we assume a "slow" access time of 100 cycles for main memory, and every cache line has four 64-bit words. These figures may be typical for a moderately loaded system with a fast clock speed of 200 MHz or greater. We assume processors with pipelined floating point performance and which support the memory model described below.

2.1. Cache Coherency

Caching has long been a popular technique to reduce the average latency of a memory system. However, caching in case of a multiprocessors becomes non-trivial when copies of the same cache line may exist in the various processors caches. The question of cache coherency addresses the problem of what should take place if one processor decides to modify one of the shared cache data items.

Cache coherency schemes are typically referred to as either hardware or software based. Hardware schemes maintain status bits for cache lines and main memory which reflect the state of data, and these bits are updated at run time by special purpose hardware. These bits are then used to determine when lines of data will be updated or invalidated during execution. Software schemes, on the other hand, make decisions regarding invalidation of cache lines based on a static compile-time evaluation of a program. Since this compile-time analysis must ensure correct program behaviour it will often result in more data being invalidated than with a hardware scheme, and hence slower performance. However, software schemes have the obvious benefit of not relying on potentially expensive special purpose hardware. Beyond these general characteristics one must have an understanding of the memory model to be supported before further defining the coherency scheme implementation. For a thorough description of cache coherency schemes see [3].

3. Weak Coherency Schemes

The goal of a weak coherency scheme is to provide the architect with more flexibility in terms of the implementation of the shared memory model. This flexibility comes at the cost of providing the user with a more difficult model to use. In particular the user must assume that the instructions in a weakly coherent memory model may be reordered, and that the completion of any instruction may be delayed indefinitely in many such models. The degree to which these events may take place is defined explicitly in the scheme, and it becomes the responsibility of the user or the compiler writer to ensure that a program will always be correct given these assumptions.

The primary benefits of a weaker scheme is that the architect is then able to use techniques such as memory access pipelining and write buffering. As mentioned above, Sequential Consistency has been the traditional model supported by shared memory multiprocessors. A slightly weaker model, *Processor Consistency*, relaxes some of the constraints such that pipelining of memory access is possible without violating the model by allowing writes to be delayed [4]. Roughly stated the model requires that from the perspective of a processor its own write instructions are not reordered, but a processor

may observe different ordering of writes which take place on two different processors.

With weaker memory models users typically need to be more careful of their use of synchronisation and locking in order to ensure correctness. Various models depend on the user labelling operations explicitly such as the acquisition and release of a lock, a competing access, and looping constructs. Release Consistency is typical of such schemes, where the release of a lock results in a partial barrier, after which it is guaranteed that all previous instructions are then completed, but where there is no guarantee that future instructions will not be issued. For a comparison of this class of models see [5].

The DEC Alpha model falls somewhere between Processor Consistency and Release Consistency in terms of its degree of strongness, and is similar to the model called simple *Weak Consistency*. In the Alpha model the user is provided with a full barrier synchronisation, the Memory-Barrier instruction (MB), which ensures that all previous instructions will have completed and that no future instructions will begin [6]. It therefore ensures that all instructions before an MB are strongly ordered with respect to the instructions after an MB. Between MBs writes and reads that do not access the same memory location may be reordered, though a read and a write to the same data from a processor will not be reordered. No labelling (beyond the use of MBs) is necessary.

In order to ensure correct program behaviour one must execute MB instructions each time a processor wants to share data with other processors. For example, an MB will typically be issued after a processor has written a flag, or when a processor wants to read a flag. The Alpha model allows both instruction reordering and write buffering between MB instructions.

4. Implementation of the Alpha Model

The primary component of an implementation of the Alpha model is the implementation of the Memory-Barrier instruction itself. With the architecture described above all other constraints will naturally be maintained, e.g. multiple accesses from a given processor to a particular memory element will not be reordered as long as we do not allow memory requests to be reordered within the memory bank queues. The primary responsibility of the MB instruction will be to ensure that caches become coherent at the point at which an MB completes. Pending writes inside the Alpha write buffer will automatically be forced out once an MB is issued.

4.1. The Hardware Memory Barrier

The hardware coherency scheme we propose is a write-invalidate scheme, where each line of data in main memory has the status of shared or exclusive. Briefly stated, the scheme works such that, if a processor wants to write to a cached value, it will first broadcast an invalidation for that line across the system bus (or through the network in the case of a directory based scheme). The status of the line in main memory is then changed to exclusive and the processor may write. We assume write-thru' to main memory also takes place, and a miss-on-write will result in a new cache line being loaded from main memory (i.e. write-allocate).

In the Alpha scheme there is no need for a processor to await acknowledgement when write access to a line has been requested. Writes may be buffered in the Alpha, and writes to main memory may end up reordered due to the pipelining implicit in our memory

architecture. However, at any time the coherency operations required to enforce strong ordering have at least been initiated, and MB only needs to ensure that they have all completed. It does this by sending a request for acknowledgment to the memory system, which will only be acknowledged when each memory bank queue has drained. Additionally a request for acknowledgment is also sent to each processor, which must be guaranteed to not overtake any invalidation instructions during transit. The processors then may send back the acknowledgement as soon as they receive the request, with the knowledge that any invalidations that were sent out earlier have been received and processed. Once the initiating processor receives acknowledgment for the MB from the remote processors as well as the memory system then processing may begin again.

4.2. The Software Memory Barrier

Software coherency schemes are based on the notion of an Epoch, a unit of computation within which data dependencies ideally do not exist. The simplest schemes invalidate all cached values at the end of each Epoch. More sophisticated software schemes use compile-time analysis to determine what data structures are modified within an Epoch and which are not. Data which is potentially written to is marked Shared/Writable and will be invalidated at the end of an Epoch.

Memory Barrier operations provide a simple mechanism for identifying Epochs as they will, by necessity, be placed such that data dependencies do not exist between them. Therefore the Alpha model simple requires that we invalidate either all cached data, or all shared/writable cached data at each MB. We must also wait for the memory bank queues to drain, as in the hardware case. Hence a substantial benefit of the Alpha model in this context is that the potentially difficult task of identifying Epochs at compile-time becomes trivial.

5. The Simulation

The experimental results presented in this paper have been generated by trace-driven discrete-event simulations. The input for the simulations consist of traces of memory accesses and arithmetic operations which were generated by annotating scientific applications, writing in Fortran. The annotations allow the trace files to be written during program execution, and the traces are parallel in the sense that each instruction has a processor number which specifies which processor is to execute the instruction. These processor numbers are generated by the annotation in a simple fine-grain manner; typically by using the induction variable of a second level do loop modulo the number of processors in the simulated machine.

The size of the part of the program which is dispatched to each processor, referred to as the *granularity*, is an important parameter. This is particularly true with weakly coherent memory models, as the frequency of synchronisation events is inversely proportional to this grain size; i.e. if processors only work on very small pieces of data then they must execute Memory-Barrier instructions regularly, and hence they will benefit little from the weak consistency scheme. We have therefore used the largest granularity that does not risk the correctness of the program execution, typically resulting in BLAS 1 or BLAS 2 size routines being dispatched to each processor. Additionally a larger grain size often results in load balancing problems amongst the processors.

5.1. The Applications

The applications we have used to generate traces are well-known scientific codes from established benchmarks, all written in Fortran. The most well know is the Linpack benchmark, a linear algebra subroutine designed to factor a dense matrix into its lower and upper triangular components. This is a particularly floating point intensive application, though the size of the loops varies from the full width of the matrix down to very small inner loops as the matrix being consider becomes smaller and smaller.

The other two codes are both parallel versions of codes taken from the Perfect Club benchmark suite [2]. The TFRD benchmark is a simulation of the behaviour of two electrons. The most computationally intensive routine, OLDA, performs integral transformations of four matrices and a transposition. Therefore there are a fairly large number of memory references per each floating point operation. The OCEAN benchmark is a fluid dynamics application which uses the spectral method, and is hence dominated by Fast Fourier Transformation (FFT) operations. This application also has a significant number of instructions which do nothing but copy data from one data structure to another.

6. Simulation Results

Typically one would expect a hardware coherence scheme to have performance better than or equal to that of a software scheme. However, due to the added cost of providing hardware support for coherency, a software scheme with slightly worse performance than a hardware scheme may actually be the better option.

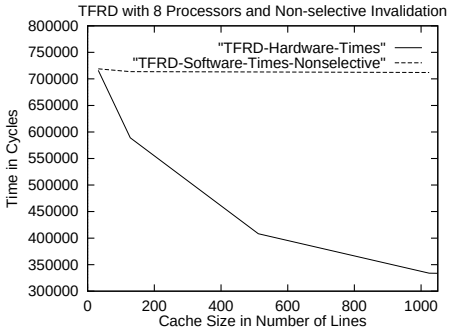

Figure 2. The Non-Selective Software Invalidation Scheme.

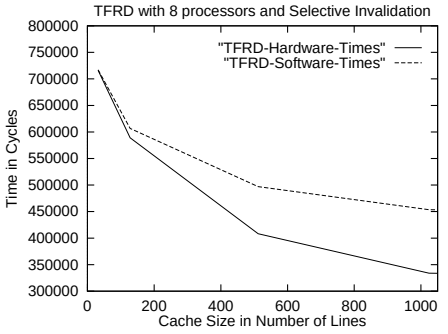

Figure 3. The Software Scheme with Selective Invalidation.

Figure 2 shows a comparison of the performance of the simplest software coherency scheme we have described, that of non-selective invalidation at each Memory-Barrier. It shows that the performance of such a scheme is only close to that of the hardware schemes at very small cache sizes. The software scheme does not benefit from cache sizes that are larger than this threshold. This makes intuitive sense if we remember that the size of an Epoch, in this case the number of operations between MBs, is fixed at a reasonably small size. Therefore the performance we can expect from a scheme with non-selective

invalidation is that all data necessary for an Epoch may fit inside the cache. Furthermore, many applications will have little data reuse within an Epoch, eg. within a doubly nested loop. The real benefit of caching comes from data which is retained in cache and reused by a processor across multiple epochs, which is clearly not possible if all data is invalidated at each MB.

In figure 3 we show the performance of the more powerful software scheme described on the TFRD application. In this scheme the program is analysed at compile time, and within an Epoch all data structures that are written to are marked as shared/writable. At the end of the Epoch all shared/writable data is invalidated, while other data is not. We see that the performance of the software scheme is roughly thirty percent slower for a cache with 4K lines, where each line has four words.

However, this performance discrepancy between the software and hardware schemes is highly dependent on the application under consideration. The primary difference between the two schemes in terms of performance is that in the software scheme all copies of a given data element are invalidated, including those in the cache of the processor writing the data. In the hardware scheme the cached data which has been written will not be invalidated, only the other cached copies of the data will be invalidated. Therefore applications where processors seldom access a given line of data after they have written to it will not see much difference between the software and hardware scheme performance. See, for example, the performance of the Linpack benchmark in figure 4. An even more substantial performance discrepancy can be seen in the OCEAN benchmark performance (see figure 5). This application uses large quantities of data, striding by one through the arrays, and there is not substantial reuse. When there is reuse it tends to be the case that the processor which has recently written to a data structure will want to read that updated value later. In the software scheme this results in a cache miss, as all caches are invalidated identically. We see that, with the four word cache lines we quickly achieve roughly a 75% hit rate, as each line loaded into cache results quickly in three cache hits, but that the Software scheme does not improve beyond this point.

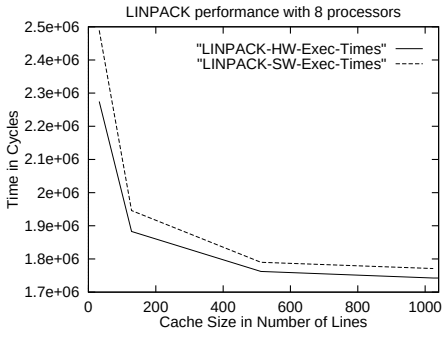

Figure 4. The Linpack Application.

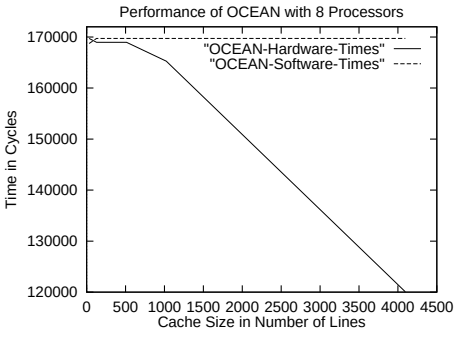

Figure 5. The OCEAN application.

We now consider the relative performance of the software and hardware schemes as a function of parallelism with a fixed size cache. In 6 we see that the performance of the

two schemes tends to converge as the number of processors is increased. This is due to the fact that, as the number of processors grows, the probability a cache line will be reused on the processor that previously wrote to it decreases.

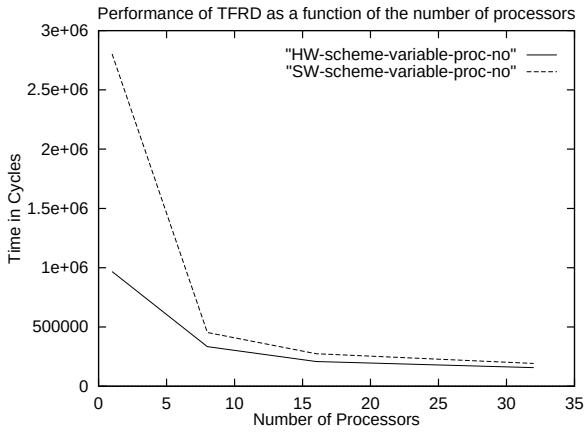

Figure 6. Performance as function of parallelism with 1000 line cache.

7. Conclusions

We have suggested possible implementations of coherency schemes which correspond to the Alpha memory model and perhaps other weakly coherent models. The software based schemes, though cheaper to implement in hardware, give performance not dissimilar from the hardware schemes for some applications, and particularly with a higher number of processors. We are currently extending this work to other underlying architectural models such as decoupled architectures.

REFERENCES

1. L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, Vol. 29, No. 9, Sept. 1979.
2. G. Cybenko, L. Kipp, L. Pointer, D. Kuck, Supercomputer Performance Evaluation and the Perfect Benchmarks", *International Conference on Supercomputing*, 1990.
3. Per Stenström, A Survey of Cache Coherence Schemes for Multiprocessors *IEEE Computer*, Vol. 23, No. 6, 1990.
4. J. Goodman, Cache Consistency and Sequential Consistency. Tech report 61, SCI Committee, March, 1989.
5. K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, M. Hill, Programming for Different Memory Consistency Models, *Journal of Parallel and Distributed Computing*, Vol. 15, 1992.
6. Alpha Architecture Handbook, Digital Equipment Corporation, 1992.